Gerenciamento Dinâmico de Memória em Aplicações com Reutilização de RDDs no Spark

Maurício Matter Donato, Rhauani Weber Aita Fazul, Patrícia Pitthan Barcelos
Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Santa Maria (UFSM)
Santa Maria, Brasil
{mdonato, rwfazul, pitthan}@inf.ufsm.br

***Resumo*—O framework Apache Spark utiliza o algoritmo LRU (Least Recently Used) para a remoção de partições de RDDs (Resilient Distributed Datasets) em caso de sobrecarga da memória. Embora suponha que partições recentemente utilizadas sejam acessadas em um futuro próximo, o LRU pode degradar o desempenho de aplicações com acessos cíclicos à memória em que a quantidade de dados manipulados excede o espaço disponível. Este trabalho apresenta o DMM (Dynamic Memory Management), um modelo de Gerenciamento Dinâmico de Memória que verifica a necessidade de remoção de partições, instrumentando a execução de aplicações e identificando o bloco a ser removido, com base na reutilização dos RDDs. Os experimentos conduzidos demonstram que o DMM pode reduzir significativamente o tempo médio de execução da aplicação quando comparado ao algoritmo LRU nativamente implementado pelo Spark, provendo assim uma melhor utilização da memória e possibilitando maior estabilidade na execução das aplicações no cluster.**

***Index Terms*—gerenciamento de memória, reutilização de dados, Apache Spark, LRU, RDD**

I. Introdução

Tradicionais *frameworks* baseados no paradigma *Map-Reduce*, como o Apache Hadoop [1], falham em oferecer abstrações para acesso à memória distribuída, tornando-os ineficientes no processamento de algoritmos com reuso de dados, tais como os utilizados em *Data Mining* e *Machine Learning* [2]. Estes algoritmos são amplamente utilizados por empresas para entender seus negócios e, consequentemente, traduzir conhecimento decisão. Visando resolver essa ineficiência, o Apache Spark [3] surge como um *framework* que, aliado ao *Resilient Distributed Dataset* (RDD), implementa um mecanismo de execução multiestágio, em que os dados intermediários podem ser escritos diretamente na memória principal, dispensando a necessidade de persistência em disco.

Um RDD é uma coleção imutável de objetos que pode ser mantida em *cache* na memória principal, possibilitando sua reutilização sem necessidade de recomputação. Entretanto, conforme novos RDDs são armazenados e processados, a memória disponível tende a ficar esgotada. Nessas situações, o Spark remove as partições mantidas em *cache* de acordo com o algoritmo LRU (*Least Recently Used*) [2].

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. O segundo autor obteve auxílio do CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil.

O LRU presume que as partições recentemente utilizadas tendem a ser acessadas novamente em um futuro próximo. Embora o algoritmo seja de fácil implementação e apresente bons resultados no gerenciamento de memória, há situações em que pode haver uma degradação no desempenho. Esse comportamento pode ser observado em acessos cíclicos à memória nos quais a quantidade de dados manipulados é maior que o espaço disponível. Nesses casos, o LRU remove o bloco que será acessado em um futuro próximo, uma vez que esses foram os blocos acessados há mais tempo [4].

Um dos objetivos do Spark é realizar a computação de aplicações em memória principal, dispensando a necessidade de escrever resultados intermediários em disco. Como consequência, obtém-se um desempenho superior no processamento de aplicações em que há reutilização de dados em iterações futuras. Embora haja um ganho de desempenho nessa classe de aplicações quando comparado a *frameworks* que escrevem dados intermediários em disco, o comportamento do acesso aos dados pode ser semelhante ao acesso cíclico à memória, evidenciando um dos problemas da utilização de algoritmos como o LRU.

Este trabalho apresenta o DMM (*Dynamic Memory Management*), um modelo de Gerenciamento Dinâmico de Memória para o Apache Spark, voltado para aplicações com reutilização de dados. Por meio desse modelo, busca-se extrair métricas da execução da aplicação e dos RDDs, de modo a utilizá-las para decidir quando e quais partições devem ser removidas da *cache*. O modelo DMM consiste em dois componentes: um *Gerente de Partições* e um *Agente de Monitoramento*.

O *Gerente de Partições* coordena as partições de RDDs armazenadas em *cache* a fim de decidir quais blocos devem ser removidos em caso de sobrecarga. Para tanto, o Gerente combina a localidade temporal do LRU à Frequência de Reutilização de cada RDD manipulado na aplicação Spark. O *Agente de Monitoramento*, por sua vez, visa prever a necessidade de remoção de partições da memória por meio da instrumentação da execução de aplicações Spark.

Uma vez detectada a necessidade de remover dados da *cache*, o Agente manipula o *znode* associado ao nodo Spark cuja memória encontra-se sobrecarregada para que esse execute as rotinas de remoção de dados. *Znodes* são estruturas organizadas sob a forma de árvore, implementadas pelo *Apache ZooKeeper*, um *framework open source* com funcionalidades

de coordenação confiável para auxiliar no desenvolvimento de aplicações distribuídas [5]. Para viabilizar a instrumentação da execução das aplicações, o *Agente de Monitoramento* também faz uso de *watches* do *ZooKeeper*, os quais são responsáveis por realizar notificações em caso de alterações em *znodes*.

O modelo de Gerenciamento Dinâmico de Memória distingue-se das soluções propostas na literatura por se preocupar em antecipar a necessidade de remoção de blocos de memória, baseando-se na possibilidade de reutilização de cada bloco. De forma a validar o modelo DMM com diferentes cenários de reutilização de dados, foram realizados experimentos com *benchmarks* baseados nos seguintes algoritmos: *K-Means*, *Logistic Regression* e *PageRank*.

O artigo encontra-se organizado da seguinte maneira: a Seção II-A apresenta os *frameworks* Apache Spark e Apache ZooKeeper. A Seção III elenca os principais trabalhos relacionados encontrados na literatura. A Seção IV detalha o modelo de Gerenciamento Dinâmico de Memória proposto. A Seção V dedica-se a explanar os experimentos conduzidos e os resultados obtidos neste trabalho. Por fim, a Seção VI apresenta as considerações finais e aponta trabalhos futuros.

II. Referencial Teórico

Esta seção descreve os tópicos fundamentais acerca dos *frameworks* utilizados para o desenvolvimento da solução proposta neste trabalho. Para tanto, a Seção II-A apresenta o *framework* Apache Spark juntamente com o modelo de memória implementado nativamente. Já a Seção II-B é dedicada à apresentação do *framework* Apache ZooKeeper.

A. Apache Spark

O Apache Spark é um *framework open source* capaz de processar grandes quantidades de dados de forma paralela e distribuída [3]. Através do seu mecanismo de execução multiestágio em memória, o Spark estende o modelo *MapReduce* – consolidado pelo Apache Hadoop – de modo a facilitar o desenvolvimento de aplicações com reutilização de dados. O Spark favorece aplicações iterativas, uma vez que mantém resultados intermediários na memória ao invés de persisti-los no disco, beneficiando o processamento recorrente do mesmo conjunto de dados.

O Spark foi projetado para implementar um mecanismo de execução multiestágio em memória principal juntamente com sua principal abstração, o *Resilient Distributed Dataset* (ou simplesmente RDD) [2]. O processamento de diversas computações em memória, dispensando a escrita de dados intermediários em disco, torna o desempenho do Spark superior ao mecanismo baseado em disco utilizado pelo Hadoop. Para tanto, o RDD do Spark encapsula a manipulação dos dados a serem processados pela aplicação, distribuindo-os através dos nodos do *cluster* e assim permitindo sua execução em memória. O Spark armazena os dados do RDD em diferentes partições, garantindo a reorganização computacional e a otimização no processamento dos dados.

A criação de aplicações Spark é realizada através de uma *Application Program Interface* (API), que possibilita a criação e a interação com RDDs, utilizando transformações e ações [6]. Transformações são funções como *map* e *filter* que, a partir de um RDD como entrada, geram um ou mais RDDs. Já as ações são operações como *reduce* e *count*, que computam o RDD e geram um valor. A avaliação dos RDDs é realizada de forma *lazy*, ou seja, quando uma função de transformação é chamada, nenhum processamento é realizado. Entretanto, quando uma função de ação é chamada em um RDD, toda a sua computação é efetuada e um valor é retornado.

Os RDDs são imutáveis (*read-only*). Ainda que, aparentemente, seja possível modificar um RDD através de uma transformação, o resultado dessa transformação é um novo RDD, o que mantém intacto o RDD original. Internamente, um RDD é composto por [2]:

- uma lista de partições que armazenam os dados;
- uma função para computar cada partição;
- uma lista de dependência de outros RDDs;
- um particionador para RDDs chave-valor (opcional); e
- uma lista de locais onde computar cada uma das partições (opcional).

Ao ser gerado, um RDD carrega consigo a referência de um ou mais RDDs ancestrais adicionados a sua lista de dependências, gerando o grafo de dependências da aplicação. O grafo direcionado acíclico (DAG – *Directed Acyclic Graph*) de dependências entre RDDs, conhecido como *lineage* do RDD, é utilizado para computar o RDD juntamente com suas dependências. Ao ser aplicada uma ação ao RDD, esse é submetido ao escalonador do Spark, o *DAGScheduler*, com o objetivo de transformar a sua *lineage* em um plano de execução físico, composto por múltiplos estágios. Esse procedimento, denominado *job*, é executado sempre que uma ação é aplicada a um RDD. Por fim, o Spark executa os estágios do *job* através de *tasks* distribuídas em todos os nodos do *cluster*.

Para realizar o processamento de aplicações, o Spark se baseia em um modelo *master-worker* com três componentes principais [3], nomeadamente:

- *Driver Program*;
- *Cluster Manager*; e
- *Workers*.

O *Driver Program*, que executa como um nodo *master*, é responsável por hospedar o contexto da aplicação em um processo na *Java Virtual Machine* (JVM). Além disso, o *Driver Program* armazena o escalonador do Spark (*DAGScheduler*) e gerencia os *jobs* da aplicação. Durante a execução de aplicações, o *Driver* disponibiliza informações sobre o estado da execução da aplicação através de uma API REST (*Representational State Transfer*), a qual possibilita o monitoramento das aplicações em execução no *cluster* [7].

O *Cluster Manager* coordena as máquinas e os recursos físicos no ambiente computacional, enquanto os *Workers* são nodos que armazenam os *Executors* do Spark. Um *Executor* é um processo JVM que executa o código delegado pelo *Driver Program* e reporta seu estado. Cada *Executor* possui sua região de memória isolada, capaz de prover o armazenamento de RDDs. A gerência dessa memória é realizada pelo próprio

Executor, utilizando o algoritmo LRU. Para sincronizar os *Executors* durante a execução da aplicação, o Spark utiliza variáveis de *broadcast*, cuja função é compartilhar o estado da execução do *job* entre os nodos do *cluster*. Quando um *Executor* recebe uma variável de *broadcast*, esta variável é armazenada na mesma fração de memória em que os dados são mantidos, compartilhando o espaço com as partições de RDDs armazenadas em *cache*. Deste modo, partições de RDDs e variáveis de *broadcast* são geridas pelo algoritmo LRU implementado no Spark.

B. Apache ZooKeeper

O *framework* Apache ZooKeeper fornece serviços de coordenação confiável para ambientes distribuídos [5]. A coordenação é realizada através de um *namespace* hierárquico e compartilhado, organizado em forma de árvore. Os nodos responsáveis por manter os dados, denominados *znodes*, são estruturas similares a arquivos e diretórios, armazenados em memória principal. O *namespace* do *znode* pode conter dados associados, assim como referências a outros *znodes* (filhos). Através de *znodes*, as aplicações podem oferecer suporte a serviços de sincronização e fila de mensagens, além do gerenciamento de configuração do *cluster*.

O ZooKeeper implementa ainda o recurso de *watches*, os quais permitem que os clientes se registrem para receber notificações quando um *znode* sofrer alguma alteração. Os *watches* são atribuídos a *znodes* específicos a partir de operações realizadas pelo próprio cliente. Quando um *znode* é alterado, uma notificação é despachada para o cliente que registrou o *watch*. Com isso, é possível evitar situações de *polling*, onde clientes realizam acessos recorrentes a fim de verificar se os dados mantidos na árvore de *znodes* foram modificados [5].

III. Trabalhos Relacionados

As soluções para o gerenciamento de memória no Apache Spark comumente implementam algoritmos estáticos e reativos, os quais executam apenas quando sinalizada a necessidade de liberação de memória. Diferentemente, este trabalho apresenta um modelo para Gerenciamento Dinâmico da Memória, o qual antecipa a necessidade de remoção de dados da memória. O modelo proposto preocupa-se em identificar quais blocos de memória devem ser removidos – e quando essa remoção deve ser efetivada – com base na possibilidade de reutilização dos blocos por parte da aplicação.

Os autores em [8] apresentam o algoritmo *Weight Replacement*, que realiza o descarte de RDDs em memória de acordo com um peso, calculado a partir do custo de recomputação da partição, da frequência de utilização e do tamanho da partição. De acordo com os autores, o algoritmo obteve melhores resultados que o LRU, reduzindo o tempo de execução da aplicação *PageRank* com *datasets* de 1GB, 2GB e 4GB.

O trabalho apresentado por [9] propõe a estratégia *Least Cost Strategy* para a remoção de blocos de memória. A estratégia combina o tempo necessário para criar, remover e recomputar cada partição dos RDDs juntamente com a sua frequência de reutilização. Os autores afirmam ter obtido um desempenho 30% melhor que o LRU.

O trabalho proposto por [10] baseia-se na reutilização de partições. Neste, é apresentada a estratégia *Lowest Cost RDD*, que remove da *cache* a partição com o menor custo. O cálculo do custo considera o tempo gasto para computar a partição, a memória ocupada, o ciclo de vida do RDD e a frequência de reutilização. A validação utilizou o algoritmo *PageRank* e, segundo os autores, a estratégia permite aumentar a eficiência do *cluster*.

Os autores de [11] agregam os mecanismos de *checkpointing* e remoção de blocos, atribuindo uma prioridade de manutenção de cada RDD em memória. Essa prioridade considera os tempos de recomputação e de *checkpointing* e o grau de dependência dos RDDs. A validação considerou a memória variando entre 5GB, 10GB e 20GB e os autores relatam ganhos de desempenho de até 13,63% em relação ao LRU convencional utilizado no Spark.

Em [12] é apresentada a abordagem *MemTune*, cujo objetivo é tornar mais eficiente a repartição dos dados entre os nodos do *cluster* e o gerenciamento dos dados mantidos em *cache*. *MemTune* detecta a utilização de memória em tempo de execução e ajusta dinamicamente as porções de memória entre *cache* de dados, execução de *tasks* e operações *shuffle*. Segundo os autores, esta abordagem supera em até 46% a implementação padrão do Spark.

O trabalho de [13] propõe o algoritmo *Dynamic Setting for Memory Management* (DSMM) com o objetivo de definir o nível de armazenamento para *cache* dos dados e a fração de memória ocupada pelo Spark. O algoritmo apresentado leva em consideração o tamanho dos dados que serão mantidos em *cache*. Com base em um *threshold* estático, o algoritmo identifica se os dados serão mantidos em memória principal ou memória e armazenamento estável. De acordo com os autores, o algoritmo proposto apresenta um desempenho até 13% superior ao Spark tradicional.

IV. Modelo de Gerenciamento Dinâmico de Memória (DMM)

O modelo de Gerenciamento Dinâmico de Memória, abreviado como DMM (*Dynamic Memory Management*), visa monitorar a aplicação Spark em execução a fim de identificar, de forma antecipada, a necessidade de realizar a remoção de blocos de memória. Através do modelo proposto, busca-se diminuir a sobrecarga das operações de substituição de blocos da memória, utilizando como base a frequência de reutilização das partições. A Figura 1 exibe a estrutura do modelo DMM, o qual consiste em:

- um nodo *Spark Master*;
- n nodos *Spark Workers*; e
- um nodo executando o *Agente de Monitoramento* e o *ZooKeeper Master*.

No modelo DMM, o nodo *Spark Master* é responsável por hospedar o *Driver* da aplicação e seu contexto. Cabe ao *Driver* manter o escalonador do Spark, cuja função é transformar a *lineage* em um plano de execução composto

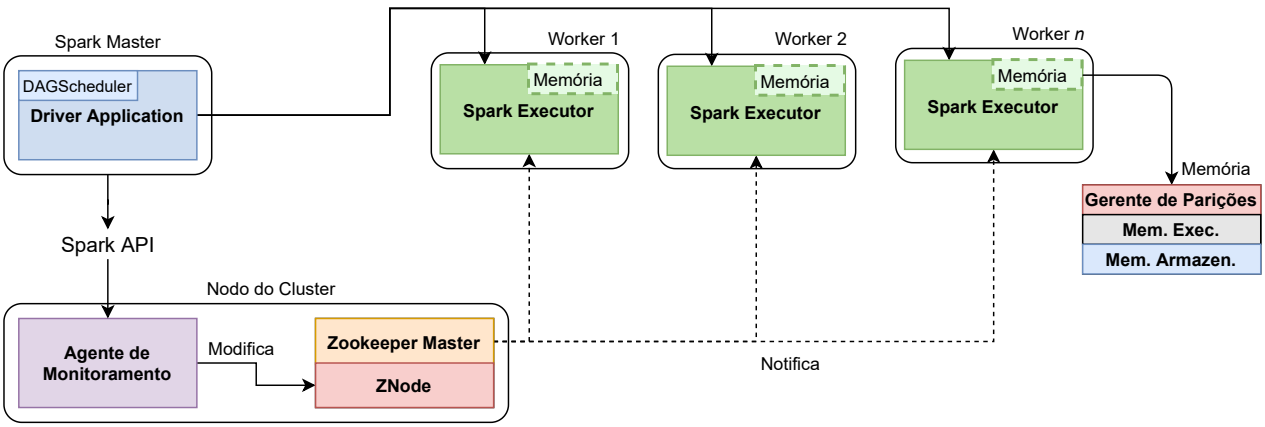

Figura 1. Modelo de Gerenciamento Dinâmico de Memória (DMM).

por estágios e gerenciar a execução desses. Ainda, o *Driver* tem como responsabilidade disponibilizar a API REST, através da qual é possível obter estatísticas relacionadas à execução. As requisições realizadas à API possibilitam identificar as aplicações em execução no *cluster*, o estado de execução de cada uma, os RDDs armazenados em *cache* e o consumo das memórias de armazenamento e de execução.

Os *Spark Workers* hospedam os *Executors*, os quais são os responsáveis por computar as aplicações Spark. Cada *Executor* é um processo JVM isolado, cuja função é executar a porção de código delegado pelo *Driver Program*. Para a implementação do DMM, os *Executors* foram alterados de modo que cada instância desse componente possui uma instância do *Gerente de Partições* para administrar a memória disponível, substituindo o LRU nativamente implementado pelo Spark. Detalhes acerca do *Gerente de Partições* são fornecidos na Seção IV-A.

O *Agente de Monitoramento*, detalhado na Seção IV-B, visa controlar e acompanhar a execução da aplicação. Já o *ZooKeeper Master* permite a sincronização e a troca de mensagens entre o *Agente de Monitoramento* e os nodos *Workers* do Spark. O ZooKeeper armazena em memória os *znodes* utilizados na implementação do DMM. Assim, os *znodes* são manipulados pelo *Agente de Monitoramento* e uma notificação é enviada aos *Executors* do Spark através do sistema de *watches* do ZooKeeper.

A. Gerente de Partições

O *Gerente de Partições* corresponde a um algoritmo que gerencia as partições dos RDDs e as variáveis de *broadcast* armazenadas na *cache*. Esse algoritmo visa priorizar os RDDs com reutilização dentro do *job*, mantendo-os na *cache* em detrimento aos demais RDDs com menor utilização. Existe um *Gerente* em cada Spark *Executor* rodando no *cluster*, de modo a efetuar o gerenciamento da memória utilizada. O *Gerente* mantém uma lista de partições ordenadas pela Frequência de Reutilização dos RDDs identificados no *job*. Em caso de empate, utiliza-se o algoritmo LRU para realizar a sub-ordenação das partições. Desta forma, garante-se que os RDDs mais frequentemente utilizados serão os últimos a serem removidos. Sendo assim, em situações em que não há reutilização de RDDs, o comportamento do DMM é similar ao do LRU nativo do Spark.

O *Gerente de Partições* não tem acesso à *lineage* da aplicação e, portanto, não consegue identificar de forma antecipada a frequência de reutilização dos RDDs manipulados no *job*. Esta responsabilidade é delegada ao *DAGScheduler*, o escalonador do Spark. A obtenção da frequência de reutilização envolve a inserção de dois métodos no *DAGScheduler*: *getLineageGraph* e *getReuseFrequency*.

O método *getLineageGraph* identifica o grafo de dependências, visando mapear todos os RDDs manipulados no *job*. O mapeamento consiste em visitar cada dependência da *lineage* do RDD cuja ação foi aplicada. Já o método *getReuseFrequency* calcula a frequência de reutilização de cada RDD identificado previamente pelo método *getLineageGraph*. Para tanto, contabiliza-se o número de vezes que um determinado RDD serviu de dependência para a criação de outro RDD na *lineage* em questão.

Os métodos inseridos no *DAGScheduler* são executados após o plano de execução ser gerado e antes do início da execução das *tasks* do *job*. Ao final da execução dos dois métodos adicionados, o resultado consiste em uma lista contendo cada RDD da aplicação com sua respectiva frequência de reutilização, a qual será utilizada pelo *Gerente de Partições*. Esta lista é mantida até o final da execução da aplicação Spark, sendo atualizada a cada novo *job* gerado pela aplicação.

A difusão da lista com os RDDs e as frequências de reutilização é realizada sob demanda entre os nodos do *cluster*. Isso significa que a comunicação entre um *Executor* e o *Driver* irá ocorrer apenas quando a memória desse *Executor* estiver sobrecarregada e necessitando remover partições. Tal difusão é realizada por meio de uma comunicação síncrona entre o *Driver* e o *Executor*, sendo realizada uma vez por *job* iniciado. No Spark, tal comunicação ocorre através de chamadas RPC (*Remote Procedure Call*), as quais podem demorar, no pior cenário, dois segundos para serem completamente executadas. Embora haja penalização no tempo de comunicação, o sincronismo é necessário, uma vez que a lista com as frequências

de reutilização dos RDDs é uma informação fundamental para realizar a ordenação das partições em memória e a remoção de dados solicitada pelo *Agente de Monitoramento*.

B. Agente de Monitoramento

O *Agente de Monitoramento* consiste em uma aplicação Java, responsável por realizar a conexão na API do Spark e efetuar o consumo dos dados da aplicação em execução. Este consumo é realizado de forma periódica, sendo determinado via parâmetro de inicialização da aplicação a periodicidade de sua execução. Utilizando os dados obtidos através da API REST, o *Agente* busca antecipar a necessidade de remoção de blocos da memória. Durante a execução de aplicações, para cada *job* são coletadas informações sobre os consumos de Memória de Armazenamento e de Memória de Execução, assim como os estágios disponíveis, juntamente com seus respectivos estados. A partir das informações coletadas, o *Agente* estima o melhor momento para realizar a liberação de memória no Spark. Para tanto, duas métricas são adotadas: o *threshold* de ocupação de memória e o estado dos estágios.

O *threshold* define um limite máximo para a ocupação da memória, de modo a manter uma porção de memória livre. Assim, nos casos em que novos dados devem ser armazenados em memória ou a Memória de Execução necessitar de espaço extra para continuar a execução da *task*, evita-se interromper a execução da aplicação para efetuar a remoção do espaço requerido. A segunda métrica consiste no estado dos estágios criados para o processamento do *job* da aplicação. Em situações em que o *threshold* foi atingido e existem estágios pendentes para execução, partições devem ser removidas da memória a fim de liberar espaço. Em contraponto, se a ocupação da memória atingir o limiar e não houver estágios pendentes, nenhuma ação é tomada. Dessa forma, permite-se que a memória seja completamente ocupada, uma vez que não é possível identificar, de forma antecipada, se a execução de um novo *job* será realizada.

Uma vez decidido que partições devem realmente ser removidas da memória, o *Agente de Monitoramento* precisa notificar os *Workers* do Spark para que esses iniciem as rotinas de remoção de blocos da memória. Essa notificação é realizada por meio do recurso de *watches* do Apache ZooKeeper. O cálculo da quantidade de memória que deve liberada, realizado de forma individual para cada *Executor*, é dado pela quantidade de memória excedente em relação ao *threshold* fixado. Na sequência, o *Agente* escreve esta informação no *znode* associado ao respectivo *Executor*.

Após a escrita dos dados no *znode*, todos os *Executors* do *cluster* com *znodes* registrados são avisados. É de responsabilidade de cada *Executor* verificar em qual *znode* o evento foi gerado. Caso o evento tenha ocorrido no *znode* de interesse do *Executor*, isto é, no nodo associado ao seu identificador único, os blocos devem ser removidos da memória. Por fim, é necessário renovar o registro no *znode* associado ao *Executor*, de forma a possibilitar o recebimento de novas notificações.

Com o modelo de gerenciamento de memória apresentado neste trabalho, a localidade temporal do LRU e a Frequência de Reutilização de cada RDD passam a ser consideradas para decidir quando e quais partições devem ser removidas ou mantidas em *cache*. A seguir, a Seção V avalia a efetividade do modelo DMM através de experimentos com diferentes cenários de reutilização de dados no Spark.

V. Experimentação e Resultados

Visando validar o comportamento do modelo proposto, foram conduzidos experimentos comparativos em um ambiente distribuído. A validação do modelo de Gerenciamento Dinâmico de Memória foi realizada na plataforma Grid'5000, utilizando um *cluster* com 8 nodos, configurados da seguinte maneira: 1 nodo Spark *Master*, 4 nodos Spark *Workers*, 1 nodo *HDFS NameNode* e *HDFS DataNode*, 1 nodo *HDFS DataNode* e 1 nodo hospedando o *Agente de Monitoramento* e o *ZooKeeper Master*.

Cada nodo do sistema era composto por dois processadores Intel Xeon E5-2630 v3 (2.4GHz, 8 *cores*/CPU), 128GB de memória RAM e 600GB de capacidade de armazenamento HDD, conectados via quatro conexões *Ethernet* de 10Gbps cada. O sistema operacional utilizado foi o Debian 8, juntamente com Java JDK 1.8.202, Apache Spark 2.2.0 e Apache Hadoop 2.7.1. Nenhum ajuste visando otimização de desempenho (*tuning*) no espaço alocado para a *heap* da JVM foi realizado, de modo a utilizar o valor padrão da plataforma.

Os experimentos foram realizados com os algoritmos *K-Means*, *Logistic Regression* e *PageRank* como *benchmarks*, implementados pelo *Intel HiBench* [14]. Esses *benchmarks* possibilitam a criação de diferentes cenários com reutilização de dados no Spark.

O *K-Means*, utilizado nos experimentos na Seção V-A, possibilita a adição e remoção de RDDs mantidos em *cache* durante a execução do *benchmark*, além da remoção de dados da memória. O *benchmark Logistic Regression*, apresentado na Seção V-B, permite a remoção de dados durante o processamento do *benchmark*, porém sem remover RDDs da *cache* no decorrer da execução. Já o *PageRank*, cujos experimentos encontram-se descritos na Seção V-C, viabilizam acesso intensivo à memória sem substituição de dados do *benchmark*.

Através destes *benchmarks*, estabelece-se três cenários com reutilização de dados no Spark, respectivamente:

- *PageRank*: acesso intensivo à memória, porém sem substituição de dados do *benchmark*;
- *Logistic Regression*: ocorrência de remoção de dados durante o processamento, mas sem realizar a remoção de RDDs da cache ao longo de toda execução; e
- *K-Means*: adição e remoção de RDDs mantidos em cache durante a execução do *benchmark*, além da remoção de dados da memória.

Para cada *benchmark*, foram criadas cinco configurações de memória total disponível: 1GB, 1,5GB, 2GB, 2,5GB e 3GB. O *Agente de Monitoramento* foi configurado para verificar o consumo de memória dos *Executors* uma vez a cada segundo, já que as *tasks* podem executar rapidamente e o espaço disponível na memória pode ser completamente ocupado. O

threshold de ocupação da memória foi adotado em 95%, visando deixar 5% da Memória de Armazenamento livre.

A opção pelo *threshold* se deu de forma experimental, utilizando o valor que obteve a maior redução no tempo de execução dos *benchmarks*. Na prática, a adoção de *thresholds* maiores de modo a reduzir a porção de memória livre tende a aumentar a quantidade de partições removidas durante a execução da aplicação. Por exemplo, em uma configuração onde há 2,5GB de memória total e utiliza-se um *threshold* de 70%, o espaço disponível para armazenamento de informações é igual a 638,61MB, sendo este valor muito semelhante à configuração de 2GB onde há 639,3MB disponíveis. Por fim, os resultados apresentados são obtidos a partir da média aritmética de 20 execuções para cada configuração de memória em cada *benchmark*.

A. Benchmark K-Means

O conjunto de dados de entrada para o *benchmark K-Means* foi de 3,7GB. Os resultados dos experimentos, ilustrados no gráfico da Figura 2, demonstram que na configuração de 1GB o DMM foi, em média, 4,46% mais rápido que o LRU. Já na configuração de 2GB de memória, o DMM obteve um desempenho, em média, 7,89% superior ao LRU. Para as configurações de 1,5GB, 2,5GB e 3GB de memória disponível, os desempenhos do DMM e do LRU foram similares.

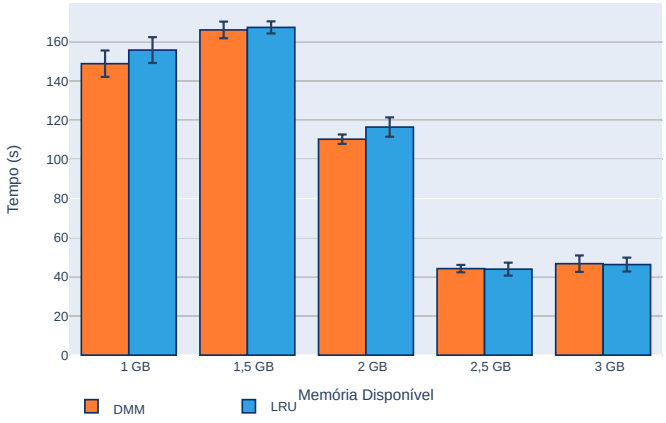

Figura 2. Tempos de Execução do *benchmark K-Means*.

Analisando os *logs* gerados pelo *K-Means*, observa-se que nas 20 execuções desse *benchmark* o processamento ocorreu em 14 *jobs*, sendo 7 *jobs* compostos por 1 estágio e os demais compostos por 2 ou 3 estágios. Em casos onde o *job* é composto por apenas um estágio, não há possibilidade de prever a existência de computações futuras. Consequentemente, o *Agente de Monitoramento* opta por não remover dados da memória, mesmo quando o *threshold* for ultrapassado.

Uma diferença entre o LRU e o DMM é a forma como os blocos de dados são removidos. Com 1GB de memória, tanto o LRU quanto o DMM removeram e inseriram uma quantidade semelhante de partições de RDDs, indicando que a memória encontrava-se sobrecarregada. Na configuração de 1,5GB, o DMM optou por remover mais variáveis de *broadcast*, mantendo uma quantidade maior de partições de RDDs em memória.

Estas variáveis de *broadcast*, as quais são utilizadas para sincronizar a execução entre os nodos do *cluster*, não possuem *lineage* e, consequentemente, possuem frequência de acesso igual a 1. Ainda, diferentemente das partições de dados, variáveis de *broadcast* possuem o nível de armazenamento padrão definido como *Memory_and_Disk*. Isso significa que, caso uma variável de *broadcast* seja requerida pelo Spark, não há o custo de retransmitir essa informação entre os nodos do *cluster*, bastando apenas recuperá-la a partir do disco de armazenamento estável.

Na configuração de 2GB de memória, observa-se a maior redução no tempo de execução quando comparado ao LRU. Analisando-se os *logs* de execução gerados, é possível perceber que na média das 20 execuções o DMM manteve mais partições em memória, removendo inicialmente as variáveis de *broadcast* utilizadas pelo Spark. Durante a execução desse *benchmark*, o LRU adicionou, em média, 37 partições de RDD a mais que o DMM. Este aumento no número de adições indica que o LRU removeu mais dados da memória, os quais seriam reutilizados. Assim, enquanto o LRU remove uma partição de RDD para inserir outra partição de RDD, o DMM remove inicialmente as variáveis de *broadcast*, para então remover partições de RDD.

Por fim, com as configurações de 2,5GB e 3GB de memória, os tempos de execução, assim como o número de operações de adição e remoção de blocos da memória, foram similares. Isso ocorre pois o espaço disponibilizado se mostrou próximo do suficiente para comportar todo o *dataset*.

B. Benchmark Logistic Regression

Para o *benchmark Logistic Regression* definiu-se um conjunto de dados de entrada com tamanho total de 7,5GB. O gráfico da Figura 3 apresenta os resultados dos experimentos com o *benchmark*, juntamente com o desvio padrão obtido para cada configuração. Na configuração de 1GB, tanto o LRU quanto o DMM não conseguiram concluir a execução da aplicação, uma vez que uma exceção de *OutOfMemoryError* foi lançada pela JVM. Esta exceção indica que não há espaço suficiente na *heap* da JVM para a alocação de um novo objeto, uma vez que o *Garbage Collector* (GC) não consegue liberar espaço para armazenar o objeto em questão e a expansão da *heap* é considerada inviável.

Quando analisamos o comportamento da execução do *benchmark*, a exceção *OutOfMemoryError* ocorre também pela alta sobrecarga no tempo de execução do GC. Esta sobrecarga indica que a JVM está gastando muito tempo na execução do coletor de lixo e pouca memória está sendo liberada. Consequentemente, esta exceção interrompe a aplicação uma vez que não há memória suficiente para dar continuidade a sua execução. Tal comportamento foi observado nas 20 execuções do *benchmark* conduzidas no experimento.

Na configuração de 1,5GB de memória, o LRU não conseguiu completar nenhuma execução do *benchmark Logistic*

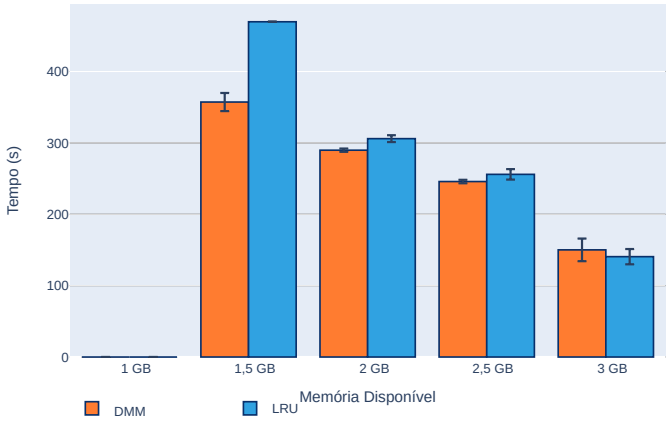

Figura 3. Tempos de Execução do *benchmark Logistic Regression*.

Regression. Assim, o tempo demonstrado para o LRU foi capturado pelo *HiBench* na execução de 29 dos 42 *jobs* da aplicação, sendo esta informação obtida através dos *logs* gerados. O DMM foi, em média, 23,94% mais rápido que o LRU, embora essa comparação não seja justa, uma vez que o LRU não concluiu o processamento da aplicação devido a falta de espaço na memória para alocação de novos objetos.

Quando dispostos 2GB de memória disponível, o DMM mostrou-se, em média, 11,82% mais eficiente que o LRU nativo do Spark. Com 2,5GB de memória disponível, o desempenho de ambos foi similar, com 3,93% de vantagem do DMM em relação ao algoritmo LRU. Já na configuração de 3GB, o DMM apresentou um desempenho, em média, 6,37% inferior ao LRU convencional. Esta degradação ocorreu devido ao *threshold* adotado, que induziu um aumento na remoção de blocos de memória a fim de evitar seu preenchimento completo. Com base nos *logs* gerados, nas configurações de 2GB e 2,5GB foi possível observar que o DMM, em média, realizou uma menor quantidade de operações de adição e remoção de partições de RDDs na memória, em comparação ao LRU. Isso demonstra que o DMM consegue priorizar os dados de RDDs, mantendo-os por mais tempo em memória. Dessa forma, o DMM opta por remover variáveis de *broadcast* antes de iniciar a remoção de dados de RDDs.

Com 3GB de memória disponível, o DMM removeu mais blocos de RDDs que o LRU devido ao *threshold* adotado. Com isso, a necessidade de manter uma fração da memória livre, imposta pelo *Agente de Monitoramento* do DMM, ocasionou a remoção de mais partições do que o necessário. Como consequência, o Spark precisou recomputar as partições perdidas, causando uma degradação média de 6,37% no desempenho do modelo DMM quando comparado ao LRU.

C. Benchmark PageRank

O experimentos realizados com o *benchmark PageRank* consideraram um conjunto de dados de entrada de 247,9MB. A Figura 4 exibe o gráfico com os resultados do *PageRank*, demonstrando os tempos de execução de cada configuração juntamente com o desvio padrão. Com a configuração de 1GB de memória disponível, o DMM foi, em média, 34,15% mais rápido que o LRU. Observando os *logs* gerados, verifica-se que o LRU introduz, de forma mais frequente, uma sobrecarga na remoção dos objetos não mais utilizados, sendo este fato demonstrado pela exceção *GC Overhead Limit Exceeded*.

A sobrecarga é causada pela implementação do LRU no Spark, que utiliza uma lista com 32 posições iniciais, a qual permite a alocação de novas posições quando atinge 75% de ocupação. Assim, a remoção de blocos nas Memórias de Execução e de Armazenamento sobrecarrega o *Garbage Collector*. Diferentemente do LRU, a implementação do DMM utiliza uma estrutura na qual não há alocação de memória prévia. Deste modo, aloca-se espaço na memória apenas quando há necessidade de armazenar novos dados.

Nas configurações com 1,5GB, 2GB, 2,5GB e 3GB de memória total disponível, os tempos de execução LRU e do DMM mantiveram-se próximos. Avaliando os *logs* de execução é possível perceber que, em ambos os métodos de gerenciamento de memória, não houve remoção de partições de RDDs da memória em nenhuma das quatro configurações de memória utilizadas. Este comportamento pode ser observado durante as 20 execuções de cada configuração.

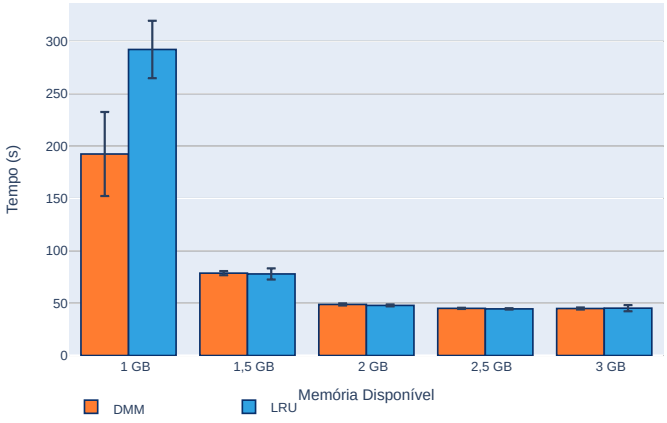

Figura 4. Tempos de Execução do *benchmark PageRank*.

Os experimentos demonstraram que a utilização do DMM pode trazer bons resultados, reduzindo em até 34,15% o tempo médio de execução de aplicações onde há reutilização de dados. Ainda, o DMM conseguiu reduzir consideravelmente a incidência de erros na execução de aplicações onde a memória encontra-se sobrecarregada. Esse fato é exemplificado na execução do *benchmark Logistic Regression* com a configuração de 1,5GB. Entretanto, há aplicações em que não é possível determinar se novas computações serão executadas, como é o caso do *benchmark K-Means*. Como consequência, torna-se inviável identificar a necessidade prévia de liberação de espaço em *cache*.

VI. Considerações Finais

O Apache Spark apresenta-se como um *framework* capaz de processar grandes quantidades de dados de maneira paralela

e distribuída. O Spark estende o modelo *MapReduce* já consolidado pelo *framework* Apache Hadoop, de modo a facilitar o desenvolvimento de aplicações com reutilização de dados. O Spark foi projetado para implementar um mecanismo de execução multiestágio em memória principal juntamente com sua principal abstração, o RDD. Porém, cabe ao desenvolvedor da aplicação Spark identificar e selecionar os RDDs que deverão ser mantidos em *cache*.

Este trabalho apresentou um modelo de Gerenciamento Dinâmico de Memória (DMM) para aplicações com reutilização de dados no Apache Spark. O modelo DMM agrega a localidade temporal do LRU à Frequência de Reutilização de cada RDD. Para tanto, o *Gerente de Partições* analisa o grafo de dependências gerado pelo *job* e identifica a frequência com que cada RDD é reutilizado. Já o *Agente de Monitoramento* supervisiona a aplicação Spark, a fim de analisar os dados obtidos de cada *Executor* do *cluster*, visando prever a necessidade de remoção de partições da memória. A decisão acerca da remoção de partições baseia-se em dois parâmetros: o *threshold* de ocupação da memória e o estado atual da execução da aplicação. Ao ser identificada a necessidade de liberação de espaço em memória, o *Agente de Monitoramento*, através do recurso de *watches* do *framework* Apache ZooKeeper, notifica o nodo Spark para que esse efetue o processo de remoção.

O modelo proposto foi validado através de experimentos, os quais possibilitaram avaliar o impacto do mesmo no tempo de execução de cada aplicação. Os experimentos foram realizados com os *benchmarks K-Means, Logistic Regression* e *PageRank*, utilizando diferentes configurações de memória disponível. Os experimentos demonstraram que o modelo DMM apresenta resultados satisfatórios, podendo alcançar desempenhos superiores ao LRU em alguns cenários, além de, em geral, prover uma melhor utilização da memória e possibilitar uma execução mais estável das aplicações.

A implementação evidenciou cenários em que o DMM apresenta melhor aproveitamento da memória, como é o caso do *benchmark PageRank*. Com esse *benchmark*, o DMM apresentou uma redução de $34{,}15\%$ no tempo médio de execução para a configuração de 1GB de memória. Já com o *Logistic Regression*, no cenário com $1{,}5$GB de memória disponível, o DMM reduziu a ocorrência de falhas durante o processamento da aplicação, possibilitando a execução do *benchmark*, ao contrário do que aconteceu com o LRU. Estes experimentos demonstraram que a solução desenvolvida apresenta resultados satisfatórios, podendo alcançar um desempenho superior ao LRU no processamento de aplicações onde há reutilização de dados. Outra característica evidenciada é o melhor aproveitamento da memória, de forma a manter mais dados em *cache* quando comparado ao LRU.

O Modelo de Gerenciamento Dinâmico mostrou ser capaz de reduzir o tempo médio de execução de aplicações com reutilização de dados. Entretanto, o algoritmo utilizado pela *Java Virtual Machine* para gerenciamento da memória pode gerar um impacto representativo nos resultados obtidos. Dessa forma, trabalhos futuros envolvem investigar o comportamento do DMM com outros algoritmos de *Garbage Collector*. Para tal, pretende-se analisar as diferenças dos algoritmos *Serial GC*, *Parallel GC*, *Concurrent Mark Sweep (CMS)* e *G1 GC* [15], bem como seus potenciais impactos no DMM.

AGRADECIMENTOS

Os experimentos apresentados neste trabalho foram conduzidos na plataforma Grid'5000, apoiada por um grupo de interesses científicos hospedado por Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais detalhes em https://www.grid5000.fr).

REFERÊNCIAS

[1] Apache Software Foundation. (2021) Apache hadoop. [Online]. Available: https://hadoop.apache.org/docs/r3.3.1/. [Acesso: Junho, 2021].

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. San Jose, CA: USENIX Association, 2012, pp. 15–28.

[3] Apache Software Foundation. (2021) Apache Spark. [Online]. Available: https://spark.apache.org. [Acesso: Junho, 2021].

[4] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 31–42. [Online]. Available: https://doi.org/10.1145/511334.511340

[5] S. Haloi, *Apache Zookeeper Essentials*, 1st ed. Birmingham: Packt Publishing Ltd, 2015.

[6] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*, 1st ed. Sebastopol: O'Reilly Media, Inc., 2015.

[7] S. Gulati, *Apache Spark 2.x for Java Developers: Explore big data at scale using Apache Spark 2.x Java APIs*, 1st ed. Birmingham: Packt Publishing Ltd, jul 2017.

[8] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, "Selection and replacement algorithms for memory performance improvement in spark," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2473–2486, 2016.

[9] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang, "Lcs: An efficient data eviction strategy for spark," *Int. J. Parallel Program.*, vol. 45, no. 6, p. 1285–1297, Dec. 2017. [Online]. Available: https://doi.org/10.1007/s10766-016-0470-1

[10] Y. Wang and T. Zhou, "A lowest cost rdd caching strategy for spark," in *Proceedings of 2019 4th International Conference on Automatic Control and Mechatronic Engineering (ACME 2019)*. CSP, 2019, pp. 30–36.

[11] M. Zhang, R. Chen, X. Zhang, Z. Feng, G. Rao, and X. Wang, "Intelligent rdd management for high performance in-memory computing in spark," in *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 2017, pp. 873–874.

[12] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 383–392.

[13] S.-J. Chae and T.-S. Chung, "Dsmm: A dynamic setting for memory management in apache spark," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 143–144.

[14] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, 2010, pp. 41–51.

[15] H. Grgic, B. Mihaljević, and A. Radovan, "Comparison of garbage collectors in java programming language," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1539–1544.